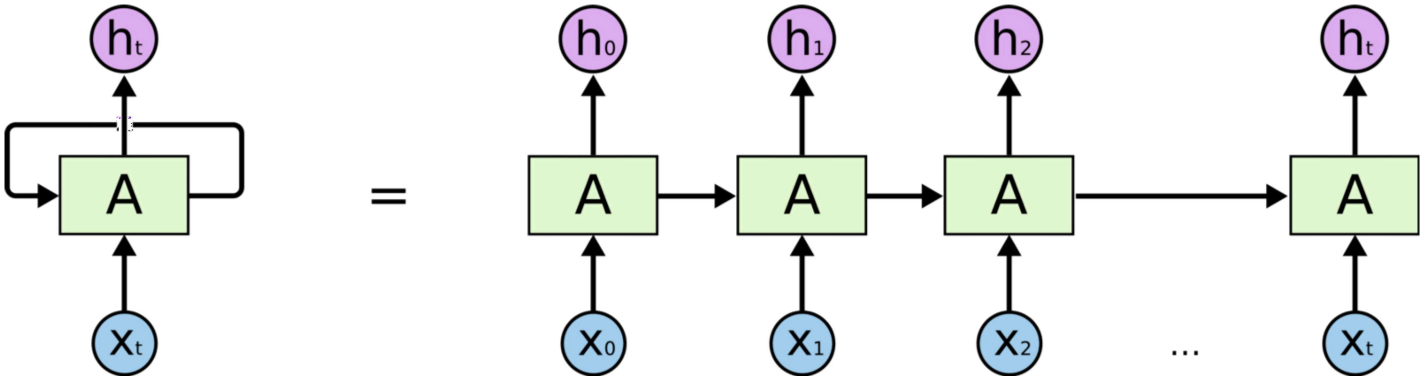
ASHRITHA K
2048029

Aim and motivation

The primary reason I have chosen to create this kernel is to practice and use RNNs for various tasks and applications. First of which is time series data. RNNs have truly changed the way sequential data is forecasted. My goal here is to create the ultimate reference for RNNs here on kaggle.

Recurrent Neural Networks

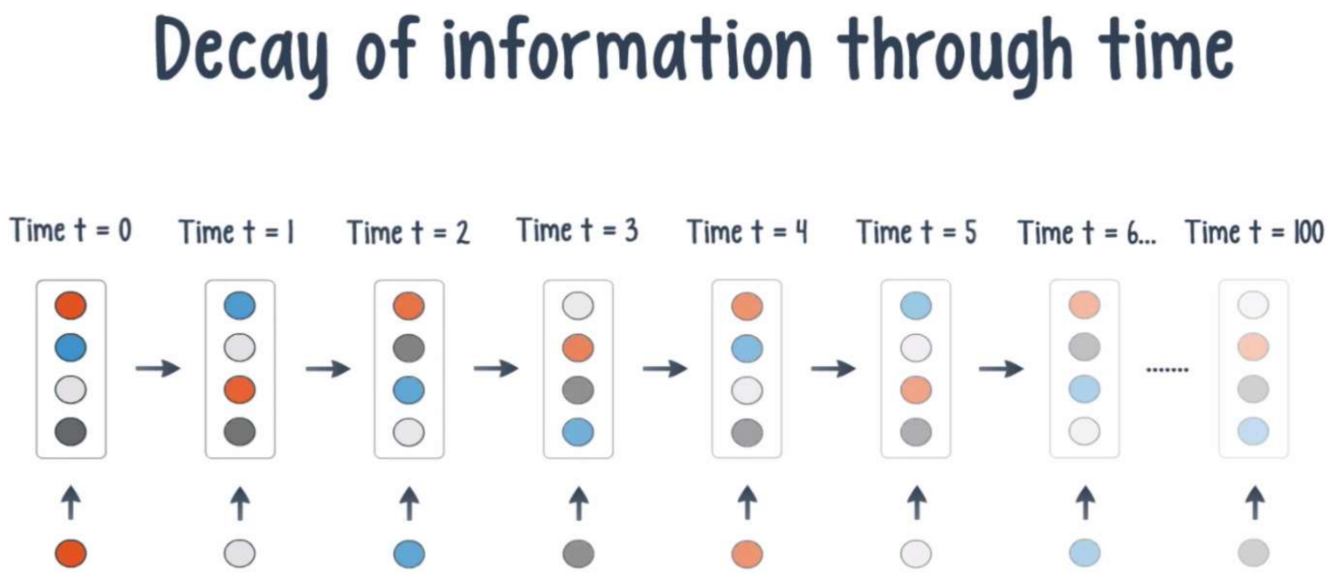
In a recurrent neural network we store the output activations from one or more of the layers of the network. Often these are hidden later activations. Then, the next time we feed an input example to the network, we include the previously-stored outputs as additional inputs. You can think of the additional inputs as being concatenated to the end of the “normal” inputs to the previous layer. For example, if a hidden layer had 10 regular input nodes and 128 hidden nodes in the layer, then it would actually have 138 total inputs (assuming you are feeding the layer’s outputs into itself à la Elman) rather than into another layer). Of course, the very first time you try to compute the output of the network you’ll need to fill in those extra 128 inputs with 0s or something.



Now, even though RNNs are quite powerful, they suffer from **Vanishing gradient problem ** which hinders them from using long term information, like they are good for storing memory 3-4 instances of past iterations but larger number of instances don't provide good results so we don't just use regular RNNs. Instead, we use a better variation of RNNs: **Long Short Term Networks(LSTM).**

What is Vanishing Gradient problem?

Vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation. In such methods, each of the neural network's weights receives an update proportional to the partial derivative of the error function with respect to the current weight in each iteration of training. The problem is that in some cases, the gradient will be vanishingly small, effectively preventing the weight from changing its value. In the worst case, this may completely stop the neural network from further training. As one example of the problem cause, traditional activation functions such as the hyperbolic tangent function have gradients in the range (0, 1), and backpropagation computes gradients by the chain rule. This has the effect of multiplying n of these small numbers to compute gradients of the "front" layers in an n-layer network, meaning that the gradient (error signal) decreases exponentially with n while the front layers train very slowly.



Long Short Term Memory(LSTM)

Long short-term memory (LSTM) units (or blocks) are a building unit for layers of a recurrent neural network (RNN). A RNN composed of LSTM units is often called an LSTM network. A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. The cell is responsible for "remembering" values over arbitrary time intervals; hence the word "memory" in LSTM. Each of the three gates can be thought of as a "conventional" artificial neuron, as in a multi-layer (or feedforward) neural network: that is, they compute an activation (using an activation function) of a weighted sum. Intuitively, they can be thought as regulators of the flow of values that goes through the connections of the LSTM; hence the denotation "gate". There are connections between these gates and the cell.

The expression long short-term refers to the fact that LSTM is a model for the short-term memory which can last for a long period of time. An LSTM is well-suited to classify, process and predict time series given time lags of unknown size and duration between important events. LSTMs were developed to deal with the exploding and vanishing gradient problem when training traditional RNNs.

Components of LSTMs

So the LSTM cell contains the following components

- Forget Gate “f” (a neural network with sigmoid)
- Candidate layer “C"(a NN with Tanh)
- Input Gate “I” (a NN with sigmoid)
- Output Gate “O”(a NN with sigmoid)
- Hidden state “H” (a vector)
- Memory state “C” (a vector)
- Inputs to the LSTM cell at any step are X_t (current input) , H_{t-1} (previous hidden state) and C_{t-1} (previous memory state).
- Outputs from the LSTM cell are H_t (current hidden state) and C_t (current memory state)

Working of gates in LSTMs

First, LSTM cell takes the previous memory state C_{t-1} and does element wise multiplication with forget gate (f) to decide if present memory state C_t. If forget gate value is 0 then previous memory state is completely forgotten else f forget gate value is 1 then previous memory state is completely passed to the cell (Remember f gate gives values between 0 and 1).

$C_t = C_{t-1} * f_t$

Calculating the new memory state:

$C_t = C_t + (I_t * C`_t)$

Now, we calculate the output:

$H_t = \tanh(C_t)$

And now we get to the code...

I will use LSTMs for predicting the price of stocks of IBM for the year 2017

```
# Importing the libraries
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('fivethirtyeight')
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import Dense, LSTM, Dropout, GRU, Bidirectional
from keras.optimizers import SGD
import math
from sklearn.metrics import mean_squared_error
```

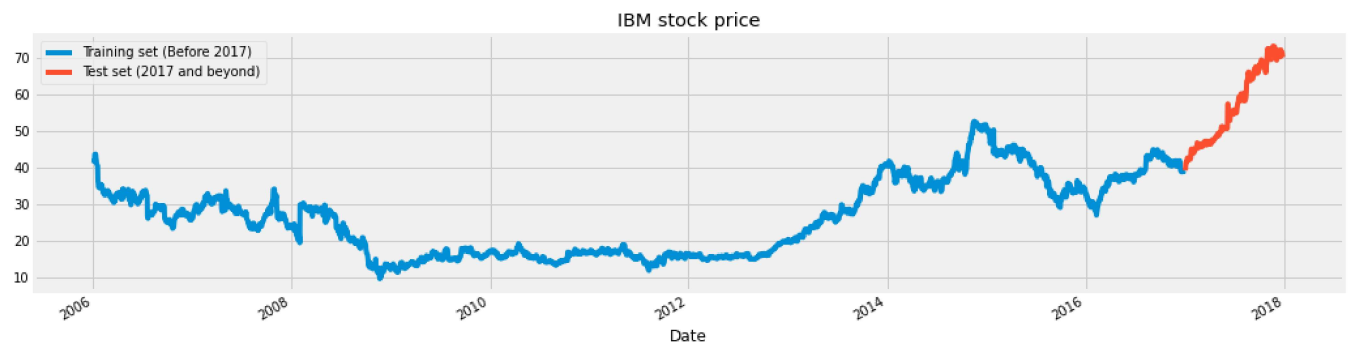
```
# Some functions to help out with
def plot_predictions(test,predicted):
    plt.plot(test, color='red',label='Real IBM Stock Price')
    plt.plot(predicted, color='blue',label='Predicted IBM Stock Price')
    plt.title('IBM Stock Price Prediction')
    plt.xlabel('Time')
    plt.ylabel('IBM Stock Price')
    plt.legend()
    plt.show()

def return_rmse(test,predicted):
    rmse = math.sqrt(mean_squared_error(test, predicted))
    print("The root mean squared error is {}.".format(rmse))
```

```
# First, we get the data
dataset = pd.read_csv('/content/AABA_2006-01-01_to_2018-01-01.csv', index_col='Date', parse_d
dataset.head()
```

Date	Open	High	Low	Close	Volume	Name
2006-01-03	39.69	41.22	38.79	40.91	24232729	AABA
2006-01-04	41.22	41.90	40.77	40.97	20553479	AABA
2006-01-05	40.93	41.73	40.85	41.53	12829610	AABA
2006-01-06	42.88	43.57	42.80	43.21	29422828	AABA
2006-01-09	43.10	43.66	42.82	43.42	16268338	AABA

```
# Checking for missing values
training_set = dataset[:'2016'].iloc[:,1:2].values
test_set = dataset['2017':].iloc[:,1:2].values
```

```
# We have chosen 'High' attribute for prices. Let's see what it looks like
dataset["High"][:'2016'].plot(figsize=(16,4),legend=True)
dataset["High"]['2017':].plot(figsize=(16,4),legend=True)
plt.legend(['Training set (Before 2017)','Test set (2017 and beyond)'])
plt.title('IBM stock price')
plt.show()
```



```
# Scaling the training set
sc = MinMaxScaler(feature_range=(0,1))
training_set_scaled = sc.fit_transform(training_set)


# Since LSTMs store long term memory state, we create a data structure with 60 timesteps and
# So for each element of training set, we have 60 previous training set elements
X_train = []
y_train = []
for i in range(60,2769):
    X_train.append(training_set_scaled[i-60:i,0])
    y_train.append(training_set_scaled[i:0])
X_train, y_train = np.array(X_train), np.array(y_train)


# Reshaping X_train for efficient modelling
X_train = np.reshape(X_train, (X_train.shape[0],X_train.shape[1],1))


# The LSTM architecture
regressor = Sequential()
# First LSTM layer with Dropout regularisation
regressor.add(LSTM(units=50, return_sequences=True, input_shape=(X_train.shape[1],1)))
regressor.add(Dropout(0.2))
# Second LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Third LSTM layer
regressor.add(LSTM(units=50, return_sequences=True))
regressor.add(Dropout(0.2))
# Fourth LSTM layer
regressor.add(LSTM(units=50))
regressor.add(Dropout(0.2))
# The output layer
regressor.add(Dense(units=1))

# Compiling the RNN
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
regressor.compile(optimizer='rmsprop',loss='mean_squared_error')
# Fitting to the training set
regressor.fit(X_train,y_train,epochs=50,batch_size=32)
```

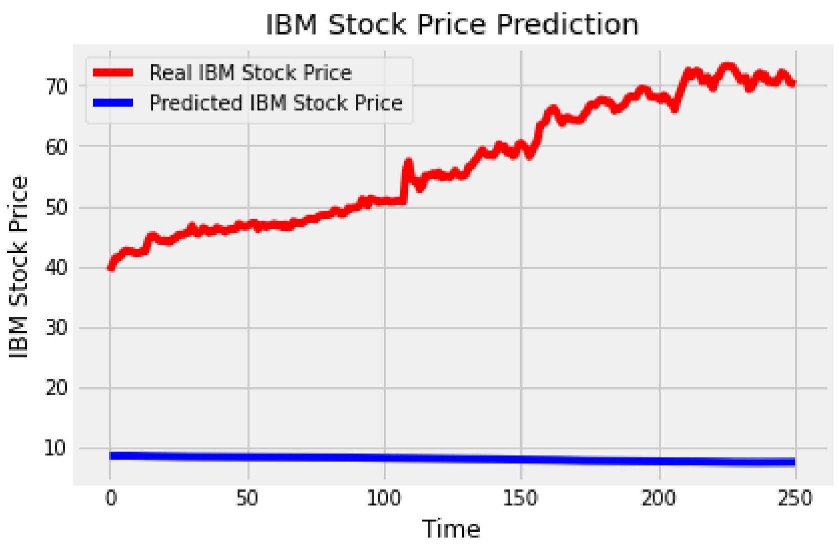
```
Epoch 1/50
85/85 [==============================] - 15s 103ms/step - loss: nan
Epoch 2/50
85/85 [==============================] - 9s 105ms/step - loss: nan
Epoch 3/50
85/85 [==============================] - 9s 106ms/step - loss: nan
Epoch 4/50
85/85 [==============================] - 9s 105ms/step - loss: nan
Epoch 5/50
85/85 [==============================] - 9s 104ms/step - loss: nan
Epoch 6/50
85/85 [==============================] - 9s 106ms/step - loss: nan
Epoch 7/50
85/85 [==============================] - 9s 105ms/step - loss: nan
Epoch 8/50
85/85 [==============================] - 9s 106ms/step - loss: nan
Epoch 9/50
85/85 [==============================] - 9s 106ms/step - loss: nan
Epoch 10/50
85/85 [==============================] - 9s 107ms/step - loss: nan
Epoch 11/50
85/85 [==============================] - 9s 107ms/step - loss: nan
Epoch 12/50
85/85 [==============================] - 9s 108ms/step - loss: nan
Epoch 13/50
85/85 [==============================] - 9s 107ms/step - loss: nan
Epoch 14/50
85/85 [==============================] - 9s 106ms/step - loss: nan
Epoch 15/50
85/85 [==============================] - 9s 107ms/step - loss: nan
Epoch 16/50
85/85 [==============================] - 9s 109ms/step - loss: nan
Epoch 17/50
85/85 [==============================] - 9s 109ms/step - loss: nan
Epoch 18/50
85/85 [==============================] - 9s 108ms/step - loss: nan
Epoch 19/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 20/50
85/85 [==============================] - 9s 110ms/step - loss: nan
Epoch 21/50
85/85 [==============================] - 9s 110ms/step - loss: nan
Epoch 22/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 23/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 24/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 25/50
85/85 [==============================] - 9s 110ms/step - loss: nan
Epoch 26/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 27/50
```

```
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 28/50
85/85 [==============================] - 9s 111ms/step - loss: nan
Epoch 29/50
85/85 [==============================] - 9s 111ms/step - loss: nan
```

```
# Now to get the test set ready in a similar way as the training set.
# The following has been done so forst 60 entires of test set have 60 previous values which i
# 'High' attribute data for processing
dataset_total = pd.concat((dataset["High"][:'2016'],dataset["High"]['2017':]),axis=0)
inputs = dataset_total[len(dataset_total)-len(test_set) - 60:].values
inputs = inputs.reshape(-1,1)
inputs  = sc.transform(inputs)
```

```
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
predicted_stock_price = regressor.predict(X_test)
predicted_stock_price = sc.inverse_transform(predicted_stock_price)
```

```
# Visualizing the results for LSTM
plot_predictions(test_set,predicted_stock_price)
```



```
# Evaluating our model
return_rmse(test_set,predicted_stock_price)
```

```
The root mean squared error is 50.16505634564676.
```

Truth be told. That's one awesome score.

LSTM is not the only kind of unit that has taken the world of Deep Learning by a storm. We have **Gated Recurrent Units(GRU)**. It's not known, which is better: GRU or LSTM becuase they have comparable performances. GRUs are easier to train than LSTMs.

Gated Recurrent Units

In simple words, the GRU unit does not have to use a memory unit to control the flow of information like the LSTM unit. It can directly makes use of the all hidden states without any control. GRUs have fewer parameters and thus may train a bit faster or need less data to generalize. But, with large data, the LSTMs with higher expressiveness may lead to better results.

They are almost similar to LSTMs except that they have two gates: reset gate and update gate. Reset gate determines how to combine new input to previous memory and update gate determines how much of the previous state to keep. Update gate in GRU is what input gate and forget gate were in LSTM. We don't have the second non linearity in GRU before calculating the outpu, .neither they have the output gate.

Source: Quora

```
# The GRU architecture
regressorGRU = Sequential()
# First GRU layer with Dropout regularisation
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activ
regressorGRU.add(Dropout(0.2))
# Second GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activ
regressorGRU.add(Dropout(0.2))
# Third GRU layer
regressorGRU.add(GRU(units=50, return_sequences=True, input_shape=(X_train.shape[1],1), activ
regressorGRU.add(Dropout(0.2))
# Fourth GRU layer
regressorGRU.add(GRU(units=50, activation='tanh'))
regressorGRU.add(Dropout(0.2))
# The output layer
regressorGRU.add(Dense(units=1))
# Compiling the RNN
regressorGRU.compile(optimizer=SGD(lr=0.01, decay=1e-7, momentum=0.9, nesterov=False),loss='m
# Fitting to the training set
regressorGRU.fit(X_train,y_train,epochs=50,batch_size=150)
```

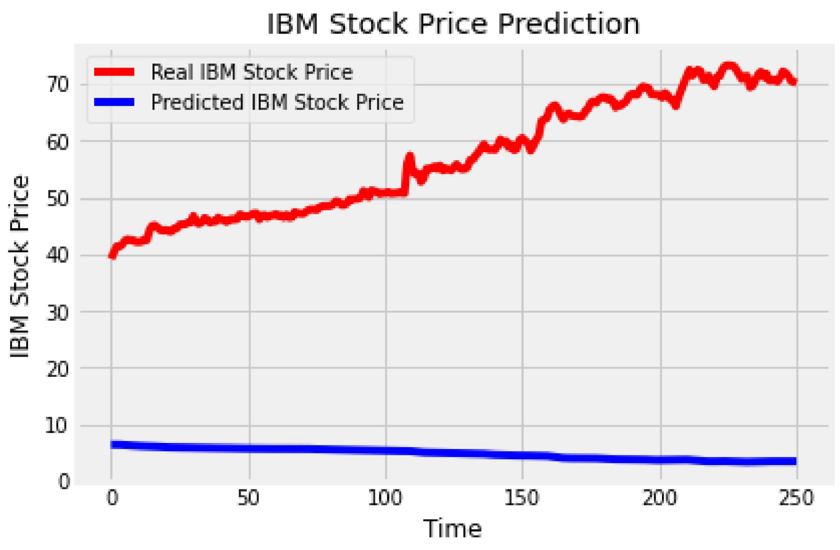
```
Epoch 1/50
19/19 [==============================] - 10s 215ms/step - loss: nan
Epoch 2/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 3/50
```

```
19/19 [==============================] - 4s 217ms/step - loss: nan
Epoch 4/50
19/19 [==============================] - 4s 217ms/step - loss: nan
Epoch 5/50
19/19 [==============================] - 4s 218ms/step - loss: nan
Epoch 6/50
19/19 [==============================] - 4s 218ms/step - loss: nan
Epoch 7/50
19/19 [==============================] - 4s 218ms/step - loss: nan
Epoch 8/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 9/50
19/19 [==============================] - 4s 223ms/step - loss: nan
Epoch 10/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 11/50
19/19 [==============================] - 4s 221ms/step - loss: nan
Epoch 12/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 13/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 14/50
19/19 [==============================] - 4s 221ms/step - loss: nan
Epoch 15/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 16/50
19/19 [==============================] - 4s 218ms/step - loss: nan
Epoch 17/50
19/19 [==============================] - 4s 221ms/step - loss: nan
Epoch 18/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 19/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 20/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 21/50
19/19 [==============================] - 4s 221ms/step - loss: nan
Epoch 22/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 23/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 24/50
19/19 [==============================] - 4s 220ms/step - loss: nan
Epoch 25/50
19/19 [==============================] - 4s 227ms/step - loss: nan
Epoch 26/50
19/19 [==============================] - 4s 229ms/step - loss: nan
Epoch 27/50
19/19 [==============================] - 4s 223ms/step - loss: nan
Epoch 28/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 29/50
19/19 [==============================] - 4s 219ms/step - loss: nan
Epoch 30/50
```

The current version version uses a dense GRU network with 100 units as opposed to the GRU network with 50 units in previous version

```
# Preparing X_test and predicting the prices
X_test = []
for i in range(60,311):
    X_test.append(inputs[i-60:i,0])
X_test = np.array(X_test)
X_test = np.reshape(X_test, (X_test.shape[0],X_test.shape[1],1))
GRU_predicted_stock_price = regressorGRU.predict(X_test)
GRU_predicted_stock_price = sc.inverse_transform(GRU_predicted_stock_price)
```

```
# Visualizing the results for GRU
plot_predictions(test_set,GRU_predicted_stock_price)
```



```
# Evaluating GRU
return_rmse(test_set,GRU_predicted_stock_price)
```

```
    The root mean squared error is 53.26597815042401.
```

Sequence Generation

Here, I will generate a sequence using just initial 60 values instead of using last 60 values for every new prediction. **Due to doubts in various comments about predictions making use of test set values, I have decided to include sequence generation.** The above models make use of test set so it is using last 60 true values for predicting the new value(I will call it a benchmark). This is why the error is so low. Strong models can bring similar results like above models for sequences too but they require more than just data which has previous values. In case of stocks, we need to know the sentiments of the market, the movement of other stocks and a lot more. So, don't expect a remotely accurate plot. The error will be great and the best I can do is generate the trend similar to the test set.

I will use GRU model for predictions. You can try this using LSTMs also. I have modified GRU model above to get the best sequence possible. I have run the model four times and two times I got error of around 8 to 9. The worst case had an error of around 11. Let's see what this iterations.

The GRU model in the previous versions is fine too. Just a little tweaking was required to get good sequences. **The main goal of this kernel is to show how to build RNN models. How you predict data and what kind of data you predict is up to you. I can't give you some 100 lines of code where you put the destination of training and test set and get world-class results. That's something you have to do yourself.**